1 Introduction

Our 9-step design process is intended for functions. Abstract data types introduce new problems, which are dealt with by means of abstraction functions and invariants. While you will have seen these ideas in COMP 15 and possibly in COMP 40, you may not have gotten the vocabulary. So this handout sketches the main ideas, with the vocabulary. It also explains how to extend and apply our design process to client code (outside of a type’s module) and implementation code (inside of a type’s module).

2 Creator, producer, observer, mutator

Following a taxonomy of Barbara Liskov, operations on an abstract data type are classified as creators, producers, observers, or mutators. This classification is explained in Programming Languages: Build, Prove, and Compare, in section 2.4.2 on page 109. It informs both the design of interfaces and the design of the client code that uses those interfaces.

3 Representation, abstraction, invariant

The new idea here is data abstraction: we have a representation that lives in the world of code, and an abstraction that lives in the world of ideas. This idea pervades COMP 40, and you may also have encountered it in COMP 15. “We can’t put a student in the computer, but we can put a representation of a student in the computer.”

For the classic data structures we study in COMP 15, the world of ideas is usually the world of mathematical ideas, like sets, sequences, and finite maps. For more practical system-building, the world of ideas is usually the outside world of problems we’re trying to solve: students, images, games, or what have you. In all cases, there are some steps you follow before you can start designing functions:

(a) Identify the abstraction.

(b) Choose the data you will use to represent the abstraction. This choice is often informed by desires about the cost model. For example, if you want constant-time lookup, you might choose a hash table.

(c) Explain the mapping from the representation to the abstraction. This mapping is called the abstraction function and is written \( A \).

(d) Design invariants that restrict the representation. Invariants sometimes also support the cost model.

In real systems, identifying the abstraction is often hard. Even experienced designers often need a lot of repetition and refinement. In COMP 105, we avoid this part of the design process—the abstractions have been designed for you.

Choosing a representation may or may not be hard. For classic data structures, you will probably find a representation in a book. For new problems, you will have to fall back on your own ideas.

Once you’ve chosen a representation, you write an abstraction function and a representation invariant. These elements play different roles:

- An abstraction function tells us what each value stands for, so we can be confident we are implementing a module’s operations according to their contracts. An abstraction function is defined only on representations that satisfy the invariant!

As an example, the abstraction function for a binary search tree usually just accumulates all the values held at all of the nodes.

The role of the abstraction function is to make sure your representation works, and to help you understand how to implement each function. The function’s contract is written in terms of the abstraction (“in the world of ideas”), but its code operates on the representation (“in the world of code”). To argue that a function’s contract is fulfilled by its implementation, you use the abstraction function to map the representations of arguments and results up to their corresponding abstractions.
A *representation invariant* tells us what is true about the representations we encounter at run time. It could be something as simple as “the list contains no duplicate elements” (for an implementation of sets as lists) or something so complicated as to demand to be written mathematically. Interesting data structures usually satisfy multiple representation invariants. These may be referred to individually, and they may also be collectively called “the invariant.”

A good example is a binary search tree. A binary search tree always has an *order invariant*—usually “smaller to the left, larger to the right.” The order invariant guarantees that a search function will find an object, if present, without having to look at every node of the tree. A serious, sophisticated binary search tree also has a *balance invariant*. There are many different forms of balance invariant, but they all guarantee that search takes a number of steps that is at most logarithmic in the number of nodes in the tree.

The role of the representation invariant is to help you write the code, and often to meet expectations about costs. Every function is permitted to rely on the representation invariant, which means it may assume that every input satisfies the invariant. And every function is obligated to guarantee the representation invariant, which means it must ensure that every output satisfies the invariant. This combination is called rely/guarantee reasoning.

These ideas will be clearer with some more examples.

## 4 Examples

### Sets

My first example abstraction is a set. To write my abstraction function, I use set notations like \{ · · · \} and \cup\cup. Potential representations include a list, a sorted list, and a binary search tree, as shown in Table 1. Any of these representations will work, but they have different cost models:

- A plain list is easy to implement, and as long as sets are small, it’s cheap. By avoiding repeated elements, we limit worst-case costs to the cardinality of the set. The abstraction function simply converts the list to a set.
- A sorted list stands for the same abstraction as an unsorted list, and so it shares the abstraction function with the unsorted list. But the sorted list satisfies an additional invariant: it is sorted. This invariant changes the cost model: adding a new element now takes half as much expected time as with the unsorted list, as does searching for an element that’s not present.
- A binary search tree is the most difficult to implement, but if it includes a balance invariant in addition to just the order invariant, it is guaranteed to do insertion, lookup and deletion in logarithmic time, even in the worst case.

The order invariant is a great example of rely/guarantee reasoning: the *lookup* function relies on the invariant, and the *insert* function both relies on it and guarantees it.

### Priority queues

My next example abstraction is a priority queue. This abstraction is actually just a sorted list of values, with operations that provide access only to the front of the list. As shown in table 2, a priority queue can be represented as a sorted list or as a binary tree, but if you studied data structures at Tufts, you probably learned to represent it as an array, under the name “heap.” My favorite representation is the binary tree: it is easy to implement, and with the “leftist heap” invariant, it is super efficient. Regardless of the representation, my abstraction function maps the representation onto a sequence of elements.

For homework, you’ll be writing abstraction functions and representation invariants for other abstractions. Representation invariants are usually pretty easy to write down: because they operate on actual representations, they can be coded, typechecked, and tested. Abstraction functions are more challenging, because by definition, they map to the world of ideas, which might not be represented in code. Here are some suggestions for the abstraction functions for homework problems:

- For natural numbers, the world of ideas is the mathematician’s world of natural numbers, sometimes written \(\mathbb{N}\). Operators and notations that are well-defined in this world include \(0\), \(S\) (successor), and \(+\).

- For integers, the world of ideas is the mathematician’s world of integers, sometimes written \(\mathbb{Z}\). Operators and notations that are well-defined in this world include \(0\), \(+\), \(−\), and \(\cdot\) (multiplication). At need, you can also resort to div and mod.

\(^{1}\)The order invariant for a binary search tree is actually hard to write down, especially in an abstract setting where the type of a value is not specified.
### Abstraction Operations

Set
- At minimum, `empty/new, insert, delete, member?:` possibly also `empty?, size, union, inter, diff`

### Representation Invariant Abstraction Function

<table>
<thead>
<tr>
<th>Representation</th>
<th>Invariant</th>
<th>Abstraction Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>No element is repeated.</td>
<td>( \mathcal{A}(\emptyset) = { } )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \mathcal{A}(x::xs) = {x} \cup \mathcal{A}(xs) )</td>
</tr>
<tr>
<td>Sorted list</td>
<td>No element is repeated; elements are sorted.</td>
<td>(Same as list.)</td>
</tr>
<tr>
<td>Binary search tree</td>
<td>No element is repeated; smaller elements are in left subtrees; larger elements are in right subtrees; perhaps some sort of balance invariant.</td>
<td>( \mathcal{A}(\text{EMPTY}) = { } )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \mathcal{A}(\text{NODE}(l,x,r)) = \mathcal{A}(l) \cup {x} \cup \mathcal{A}(r) )</td>
</tr>
</tbody>
</table>

Table 1: Representations of sets

### Abstraction Operations

Priority queue
- At minimum, `empty/new, insert, empty?, and delete-min;` possibly also `size, find-min, merge`

### Representation Invariant Abstraction Function

<table>
<thead>
<tr>
<th>Representation</th>
<th>Invariant</th>
<th>Abstraction Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>List is sorted with the smallest element at the front (inefficient unless small).</td>
<td>( \mathcal{A}(xs) = xs )</td>
</tr>
<tr>
<td>Array</td>
<td>Element at index ( i ) is not larger than the elements at indices ( 2i ) and ( 2i + 1 ), if any.</td>
<td>( \mathcal{A}(a) = \text{sort}(a) )</td>
</tr>
<tr>
<td>Binary tree</td>
<td>Element at node is not larger than elements at left and right child, if any.</td>
<td>( \mathcal{A}(\text{EMPTY}) = [] )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>( \mathcal{A}(\text{NODE}(l,x,r)) = x :: \text{merge}(\mathcal{A}(l), \mathcal{A}(r)) )</td>
</tr>
<tr>
<td>Leftist heap</td>
<td>Binary tree, with the additional invariant that every left subtree is at least as high as the corresponding right subtree.</td>
<td>(Same as binary tree.)</td>
</tr>
</tbody>
</table>

Table 2: Representations of priority queues
For coins, the simplest abstraction is probably a list of denominations, such as “quarter, quarter, dime, quarter, nickel.” Operators and notations that are well-defined in this world include all the usual operations on lists.

- For players, the abstractions are X and O.

5 How design steps are affected

Abstract data types affect the design process because of two changes:

- Contracts for functions are written for the abstraction (“in the world of ideas”), but the functions themselves are written for the representation (“in the world of code”).
- Outside its defining module, an abstract type has no forms of data.

The implications are explored below.

Design steps for client code

Outside an abstract type’s module, code is called a client of the module. Here’s how each step of the design process works in client code:

1. **Forms of data.** Manifest types have exposed representations, so client code works with them just as usual.

   Abstract types don’t give access to the forms of data. If you’re a client and you’re consuming abstract data, you’ll typically be calling observers—sometimes mutators. And to make new abstract values, you’ll use creators and producers.

   A values of abstract type is a little bit analogous to a function: all you can do with it is what’s in the interface.

2. **Example inputs.** Example inputs can be made only by calling creators, producers, and sometimes mutators. (Mutation makes testing harder.)

3. **Function’s names.** Nothing changes.

4. **Function’s contracts.** The client’s own contracts may mention the abstraction or may even be written in terms of the abstraction. The client doesn’t know about the representation—which is mostly the point.

5. **Example results.** Any example results of abstract type have to be expressed indirectly, again using creators, producers, and sometimes mutators. (Mutation makes testing harder.)

6. **Algebraic laws.** When client code consumes a value of abstract type, it can’t have one algebraic law per form of data—an abstract type has only one form of data. Instead, to decide on a law, you can use side conditions from an observer, or you can even break an observed value down by forms of its data. For example, in the Abstract Game Solver, it’s useful to write laws for the advice function by breaking down the list of legal moves.

7. **Case analysis.** You can’t do case analysis on a value of abstract type directly. But you may be able to do case analysis on the results of calling an observer in the interface.

8. **Coding.** Right-hand sides are coded as usual.

9. **Revisit unit tests.** Unit tests can be written as usual. To construct values for use in unit tests, you can call creators and producers (and possibly mutators) for the abstract type.

Design steps for implementations

Inside an abstract type’s module, code has complete access to the representation. Here’s how each step of the design process works in an implementation:

1. **Forms of data.** The implementation has complete access to the representation, so forms of data are available as usual.

2. **Example inputs.** Example inputs can be written as usual—but care must be taken to be sure that every example input satisfies the representation invariant.

3. **Function’s names.** Nothing changes.

4. **Function’s contracts.** The contract of each function is given in the interface, and it is written in terms of the abstraction. But the function itself is written in terms of the representation. To be confident that each function fulfills its contract, you must define the function with the type’s abstraction function in mind.

   Also, each exported function has these amendments added to its contract:

   - Every input of abstract type satisfies the representation invariant for that type.
   - Every output of abstract type must satisfy the representation invariant for that type.

   A function is exported if it is visible in the interface that is being implemented. C++ calls these functions “public.”
Private functions may, if they wish, deal with arguments and results that don’t satisfy the representation invariant. Indeed, one common use of private functions is to re-establish an invariant before returning a result. Each private function’s relationship to and action on representation invariants must be documented in its contract.

5. *Example results.* Example results can be written as usual, as can unit tests.

6. *Algebraic laws.* Algebraic laws can be written as usual.

7. *Case analysis.* Case analysis can be based on algebraic laws as usual.

8. *Coding.* Right-hand sides are coded as usual.

9. *Revisit unit tests.* Unit tests can be written as usual.